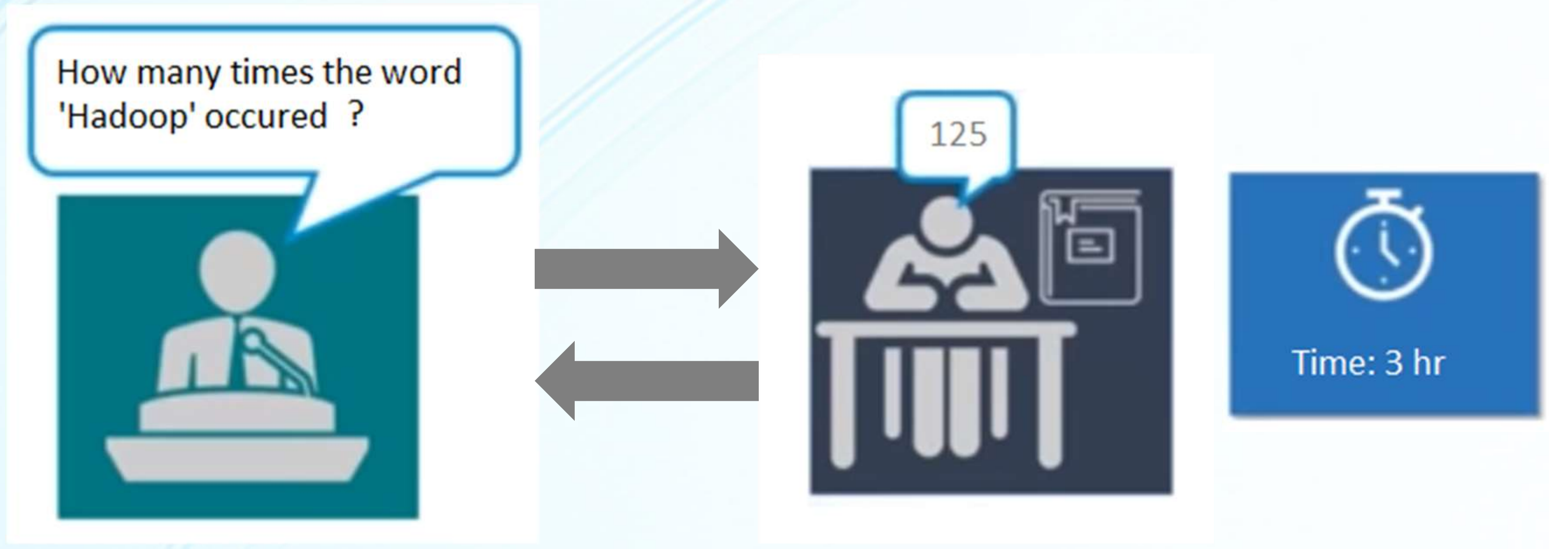
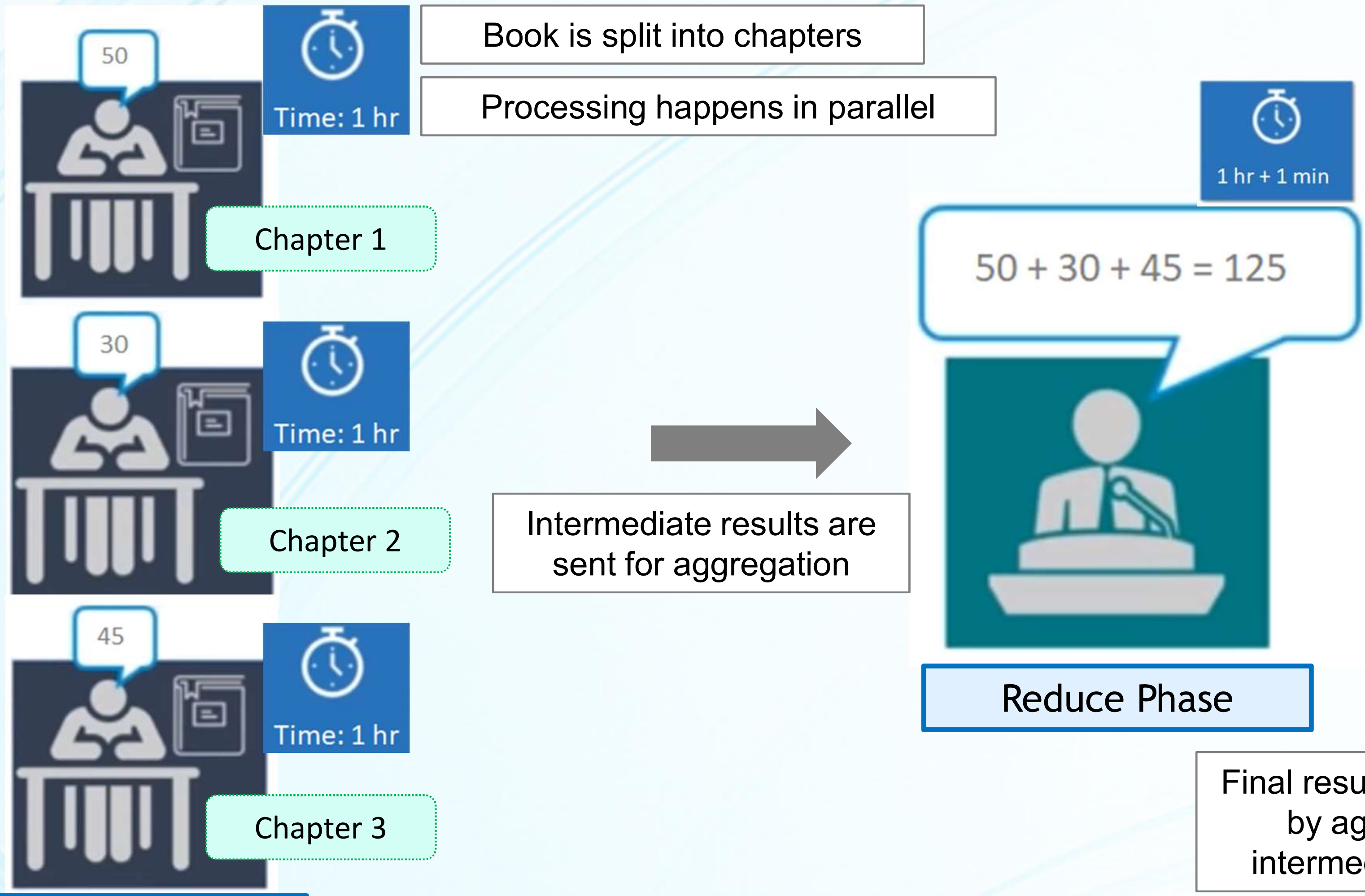
MAPREDUCE

MAP REDUCE

Let’s start with a Story



Let’s start with a Story



Book is split into chapters

Processing happens in parallel

Intermediate results are sent for aggregation

Reduce Phase

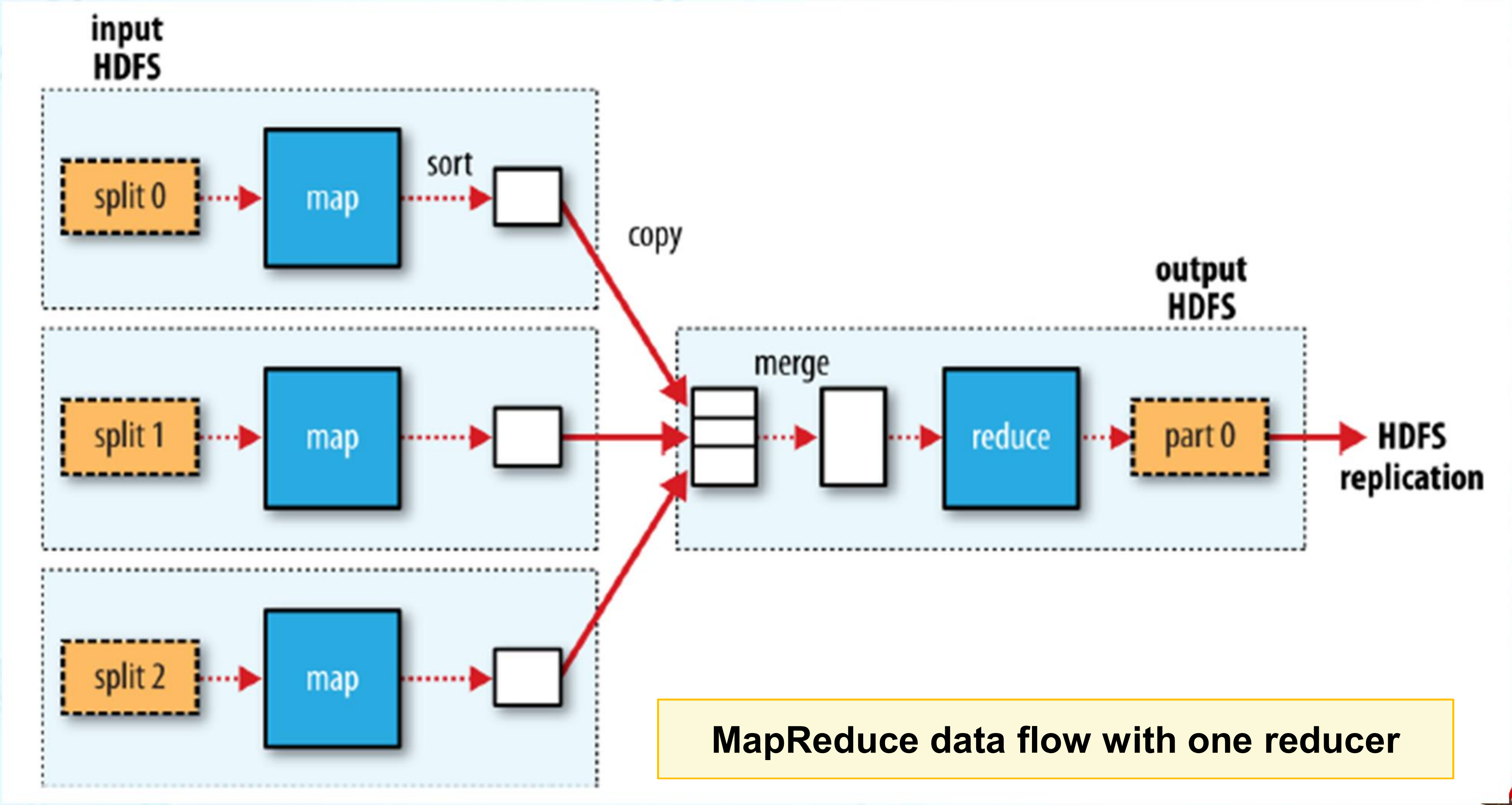
Final result is computed by aggregating intermediate results

Map Phase

What is MapReduce ?

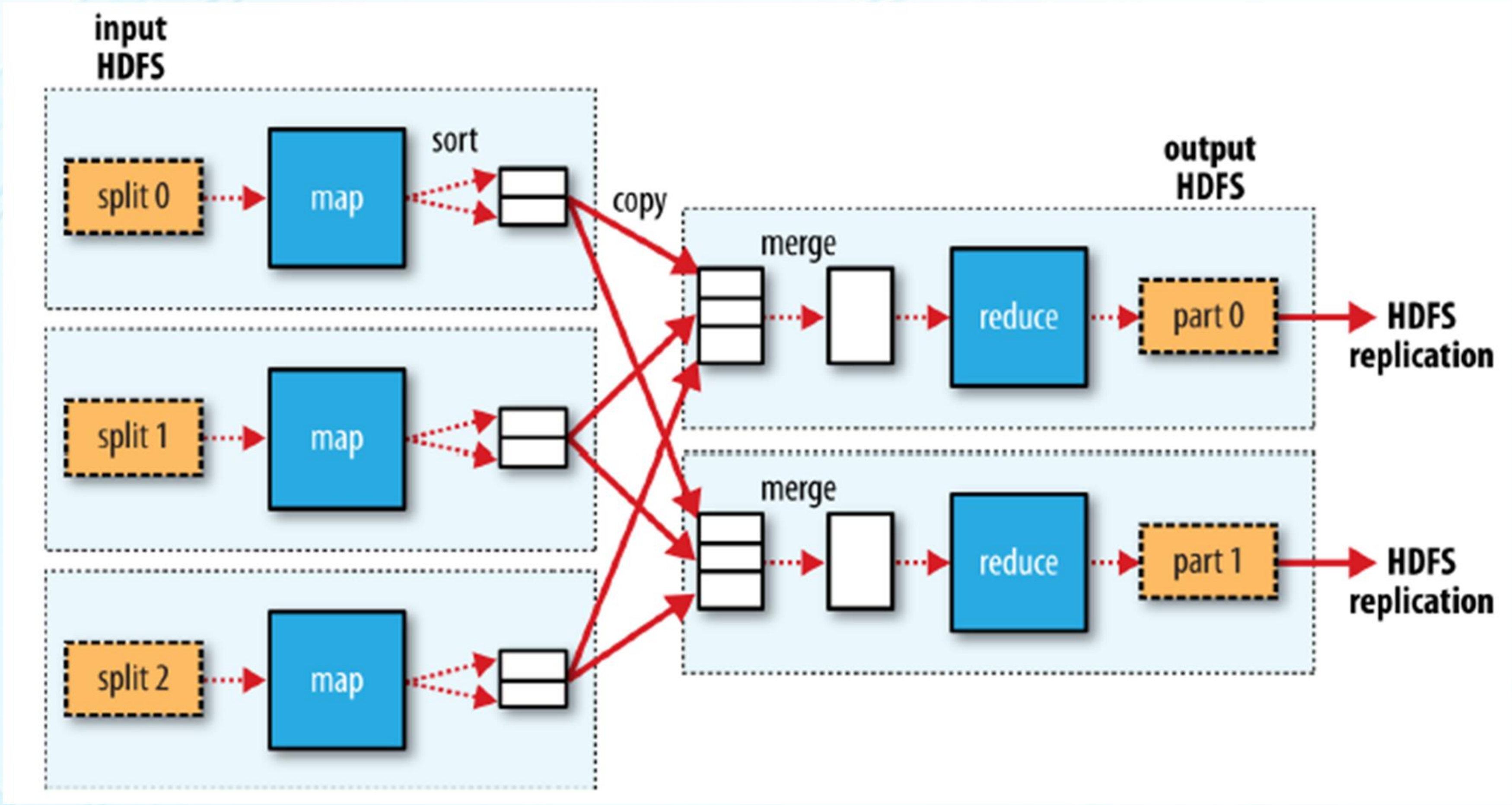
What is MapReduce ?

MapReduce is a **programming framework** for performing **distributed** and **parallel** processing on large datasets using the **MapReduce programming paradigm**.



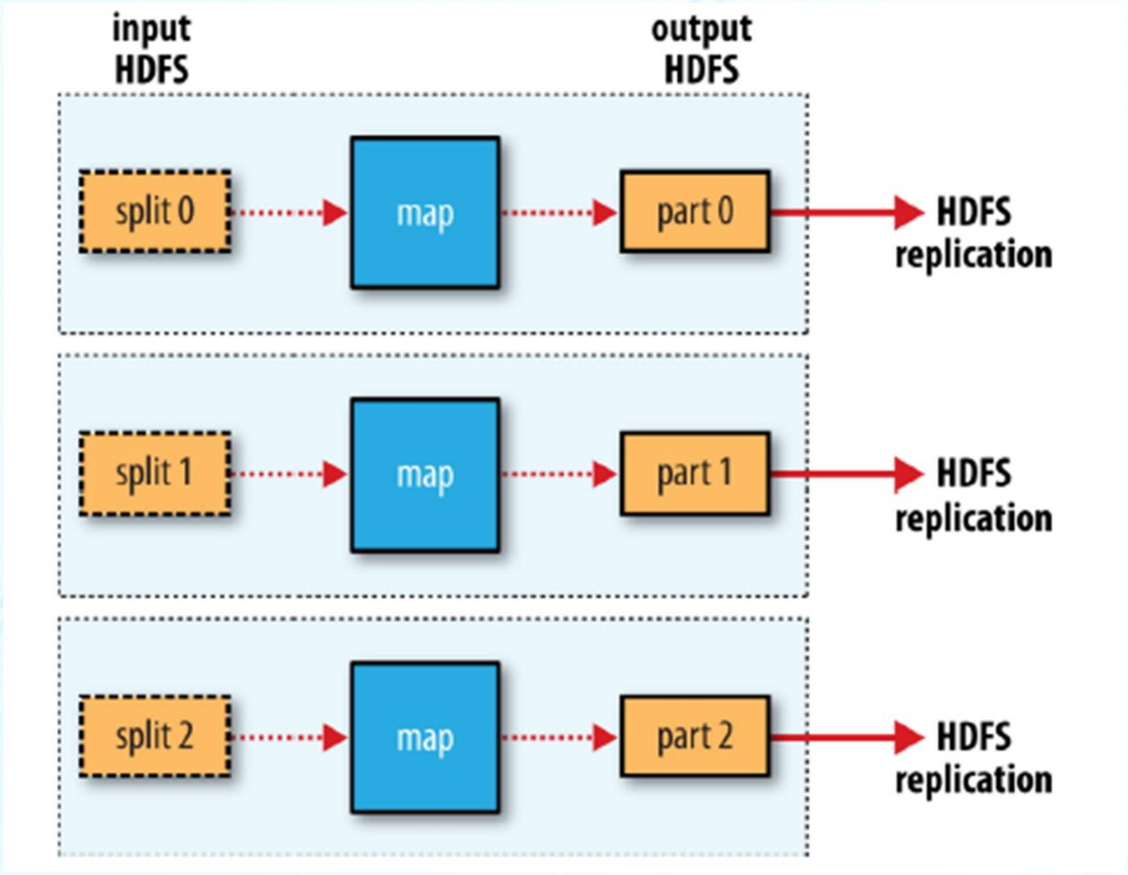
MapReduce data flow with one reducer

What is MapReduce ?



MapReduce data flow with multiple reducers

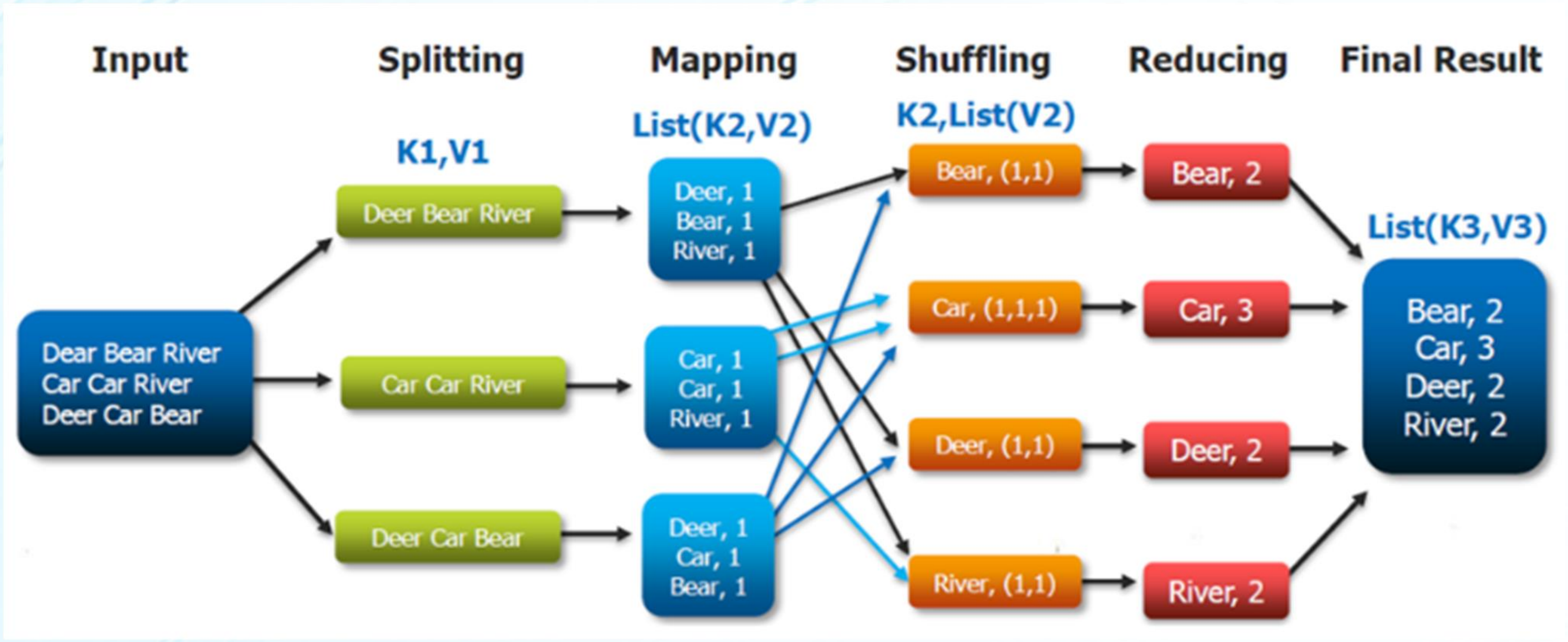
What is MapReduce ?



MapReduce data flow with no reducers

Let’s understand with an example

Word Count Program



MR Classes

The **Driver class** allows the user to configure the job, submit it, control its execution, and query the state. You specify all Job specific configurations in this class such as Job Name, Input file Path, Output File Path, Mapper class name, Reducer class name etc.

The **Mapper class** defines the Map job. Maps input key-value pairs to a set of intermediate key-value pairs. Maps are the individual tasks that transform the input records into intermediate records.

The **Reducer class** defines the Reduce job in MapReduce. It reduces a set of intermediate values that share a key to a smaller set of values.

The **Combiner class** works as 'semi-reducer' and is used to summarize the map output records with the same key. The output of the combiner will be sent over the network to the actual Reducer task as input.

The **Partitioner class** allows to control which type of data goes to which reducer. We can implement our own custom partitioning strategy

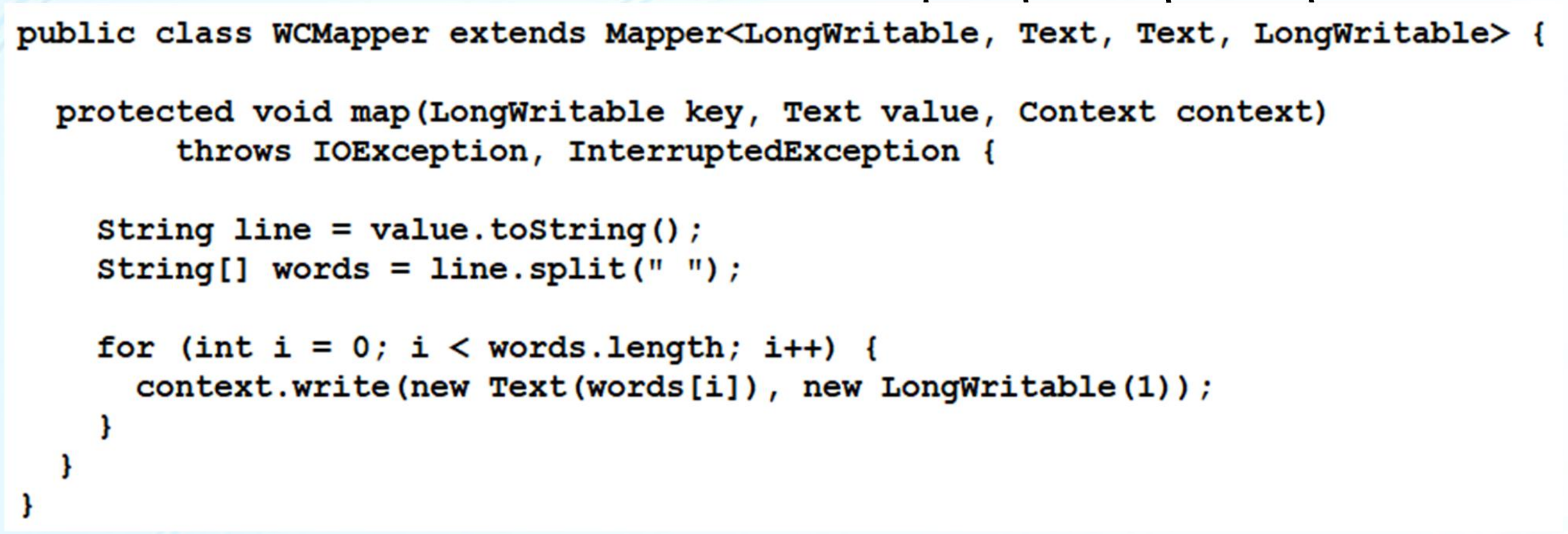
Mapper Code

Input Key Type

Input Value Type

Output Key Type

Output Value Type

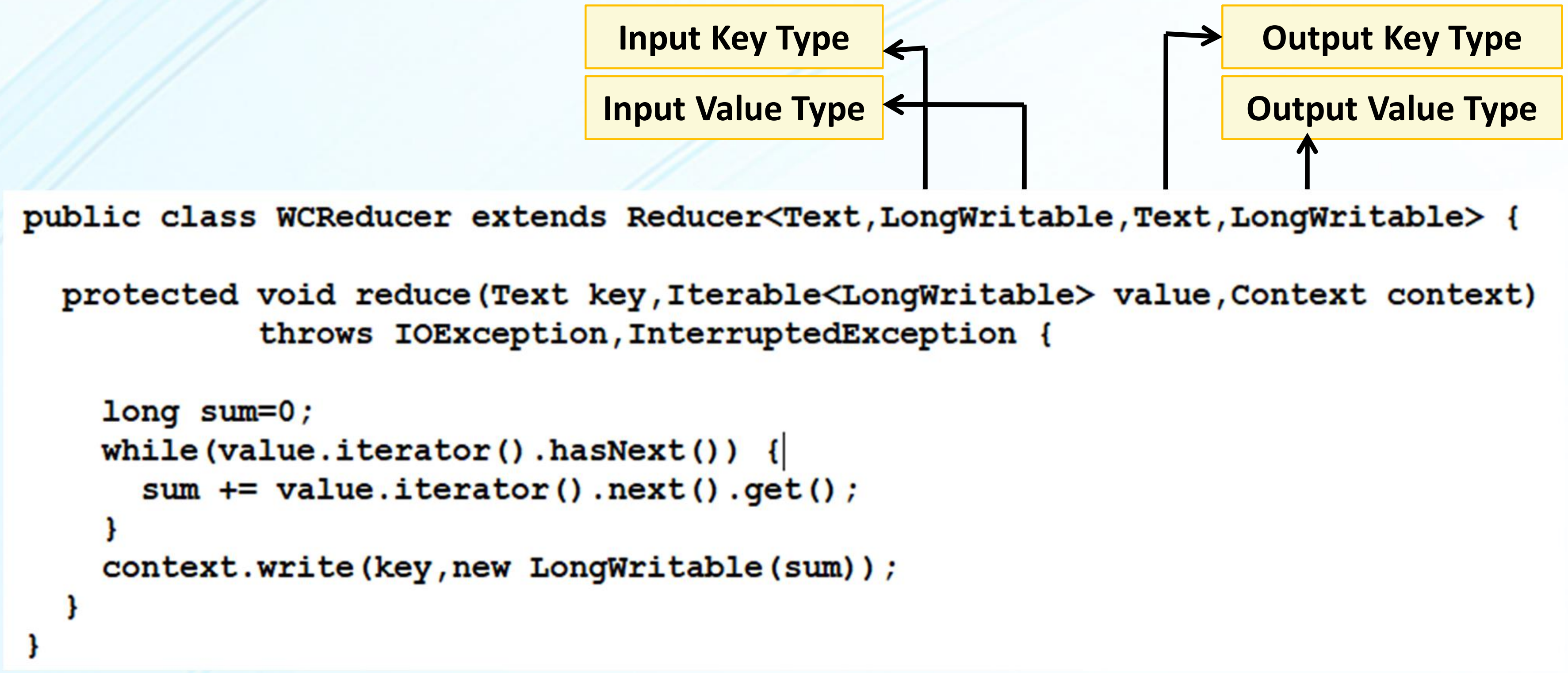


```
public class WCMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

  protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

    String line = value.toString();
    String[] words = line.split(" ");

    for (int i = 0; i < words.length; i++) {
      context.write(new Text(words[i]), new LongWritable(1));
    }
  }
}
```

Input Key: Byte Offset
Input Value: Each line

Output Key: Each Word in the line
Output Value: 1

Reducer Code



```
public class WCReducer extends Reducer<Text,LongWritable,Text,LongWritable> {

  protected void reduce(Text key,Iterable<LongWritable> value,Context context)
            throws IOException,InterruptedException {

    long sum=0;
    while(value.iterator().hasNext()) {
      sum += value.iterator().next().get();
    }
    context.write(key,new LongWritable(sum));
  }
}
```

- Keys are the unique words that are generated from the shuffle & sort phase: **Text**
- Value is a list of integers corresponding to each key: **LongWritable**

- Key is the unique word present in the input file: **Text**
- Value is total sum of word counts: **LongWritable**

Driver Code

In the Driver class, we set the configuration of the MR job to run in Hadoop

```
Job wordcountjob = new Job(getConf());

wordcountjob.setJobName("MR_WORDCOUNT");
wordcountjob.setJarByClass(this.getClass());

wordcountjob.setMapperClass(WordCountMapper.class);
wordcountjob.setReducerClass(WordCountReducer.class);

wordcountjob.setMapOutputKeyClass(Text.class);
wordcountjob.setMapOutputValueClass(LongWritable.class);
wordcountjob.setOutputKeyClass(Text.class);
wordcountjob.setOutputValueClass(LongWritable.class);

FileInputFormat.setInputPaths(wordcountjob,new Path(args[0]));
FileOutputFormat.setOutputPath(wordcountjob,new Path(args[1]));

wordcountjob.setNumReduceTasks(2);

return wordcountjob.waitForCompletion(true)==true? 0:1;
```

Job Name

Job Class

Mapper & Reducer Classes

Output K-V Types

Input & Output file paths

No. of Reducers

Let’s look at YARN

The processing framework on Hadoop

What is YARN?

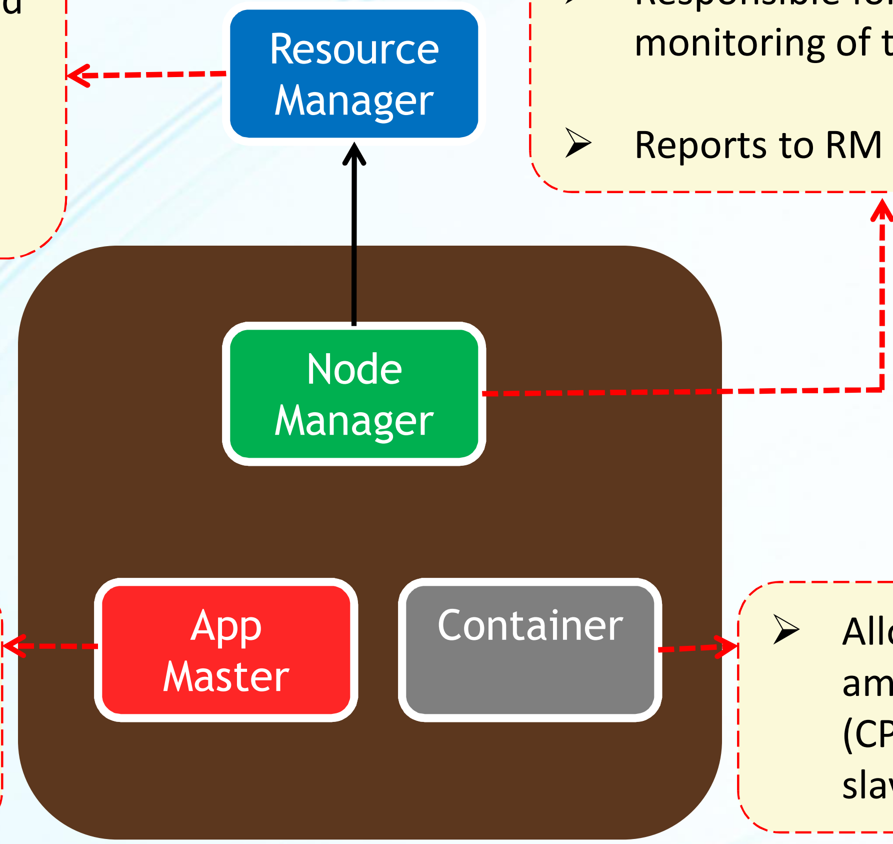
- YARN (Yet Another Resource Negotiator) is the resource management framework responsible for assigning computational resources for application execution.

- Introduced in Hadoop 2.0 as the default resource manager, replacing the old processing daemons - JobTracker and TaskTracker.

Components of YARN

- Master daemon that manages other daemons and accepts job submissions
- Allocates container for the AppMaster

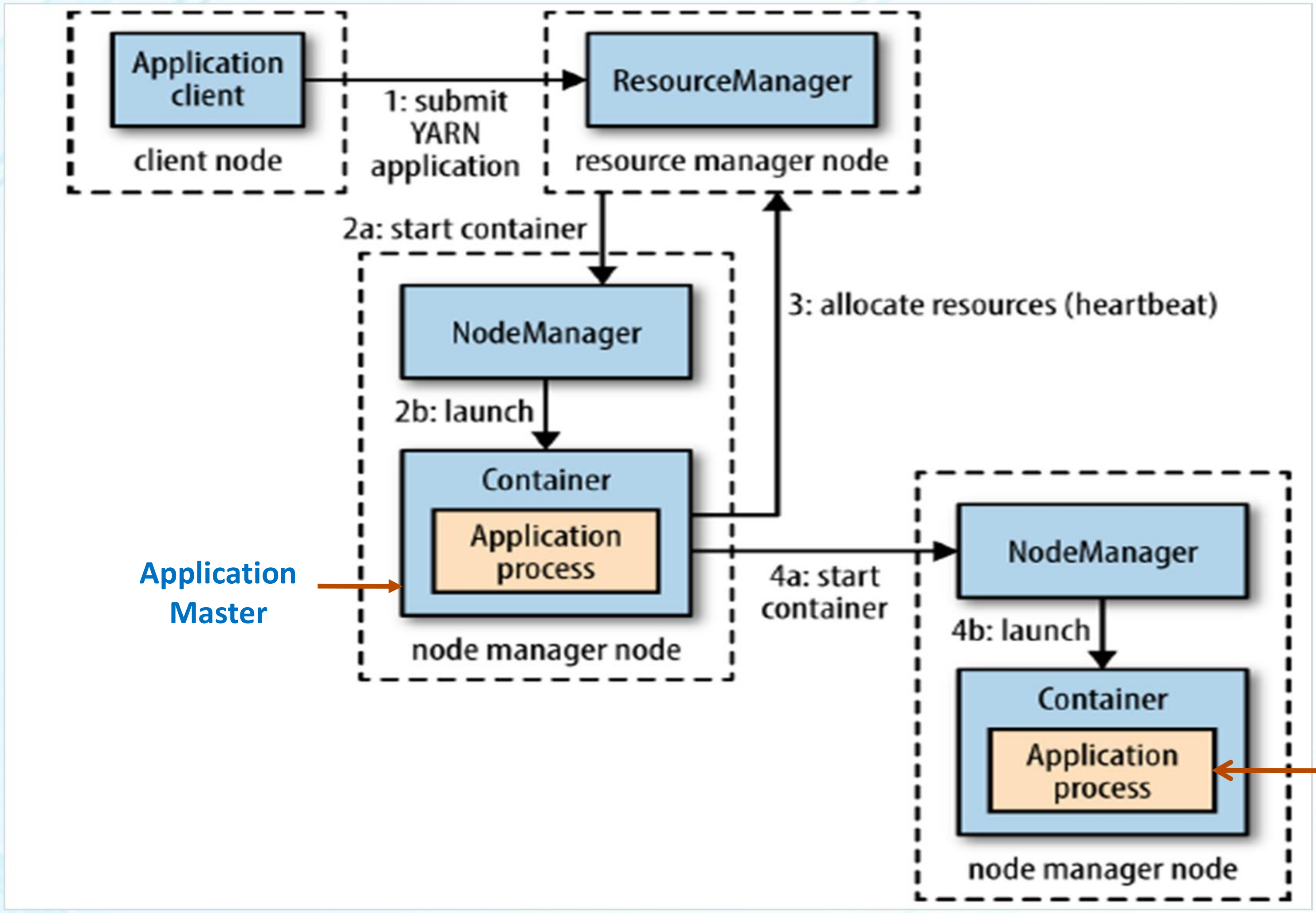
- Responsible for containers and monitoring of their resource usage
- Reports to RM through heartbeats.



- One per application
- Manages MR jobs
- Negotiates resource from RM

- Allocates certain amount of resources (CPU, RAM etc) on a slave node (NM)

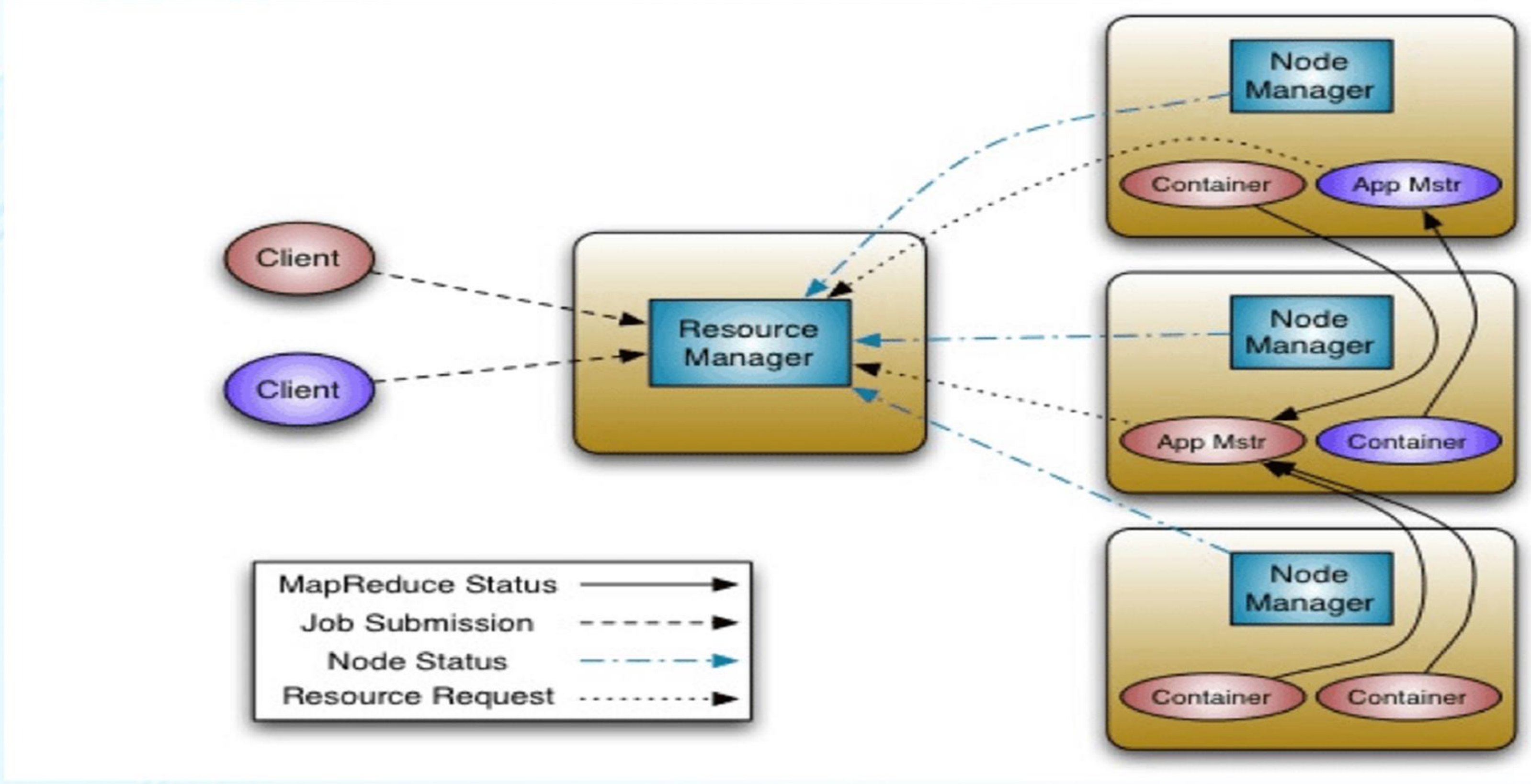
YARN Job Workflow



Application Master

MR Task

YARN Architecture



- Resource Manager (one per cluster)
- Application Master (one per application)
- Node Managers (one per node)

YARN Job Workflow

Typical application execution with YARN follows this flow:

1. Client program submits the MR application to the RM, along with information to launch the application-specific Application Master.

2. RM negotiates a container for the AM and launches the AM.

3. AM boots and registers with the RM, allowing the original calling client to interface directly with the AM.

4. AM negotiates resources (resource containers) for client application.

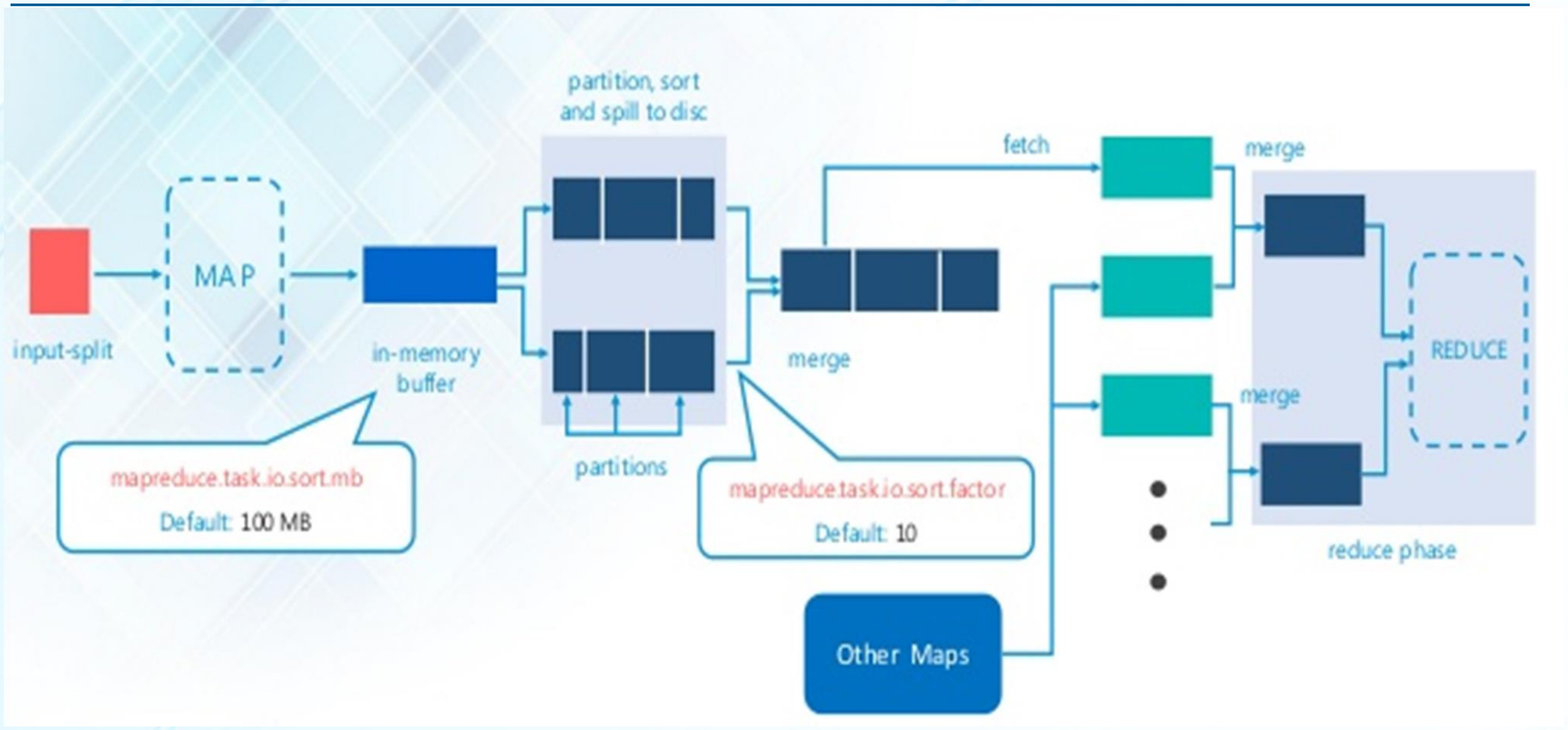
5. AM gives the container launch specification to the NM, which launches a container for the application.

6. During execution, client polls AM for application status and progress.

7. Upon completion, AM deregisters with the RM and shuts down.

Let’s have look at MapReduce again

MapReduce Job Workflow



Mapper Lifecycle

setup

- Called once at the beginning of the task. Mainly used for initializations etc.

map

- Called once for each key/value pair in the input split.
- Most applications should override this, but the default is the identity fn.

cleanup

- Called once at the end of the task.

run

- Override this method for more complete control over the execution of the Mapper.

Reducer Lifecycle

setup

- Called once at the beginning of the task. Mainly used for initializations etc.

reducer

- Called once for each key/value pair in the input split.
- Most applications should override this, but the default is the identity fn.

cleanup

- Called once at the end of the task.

run

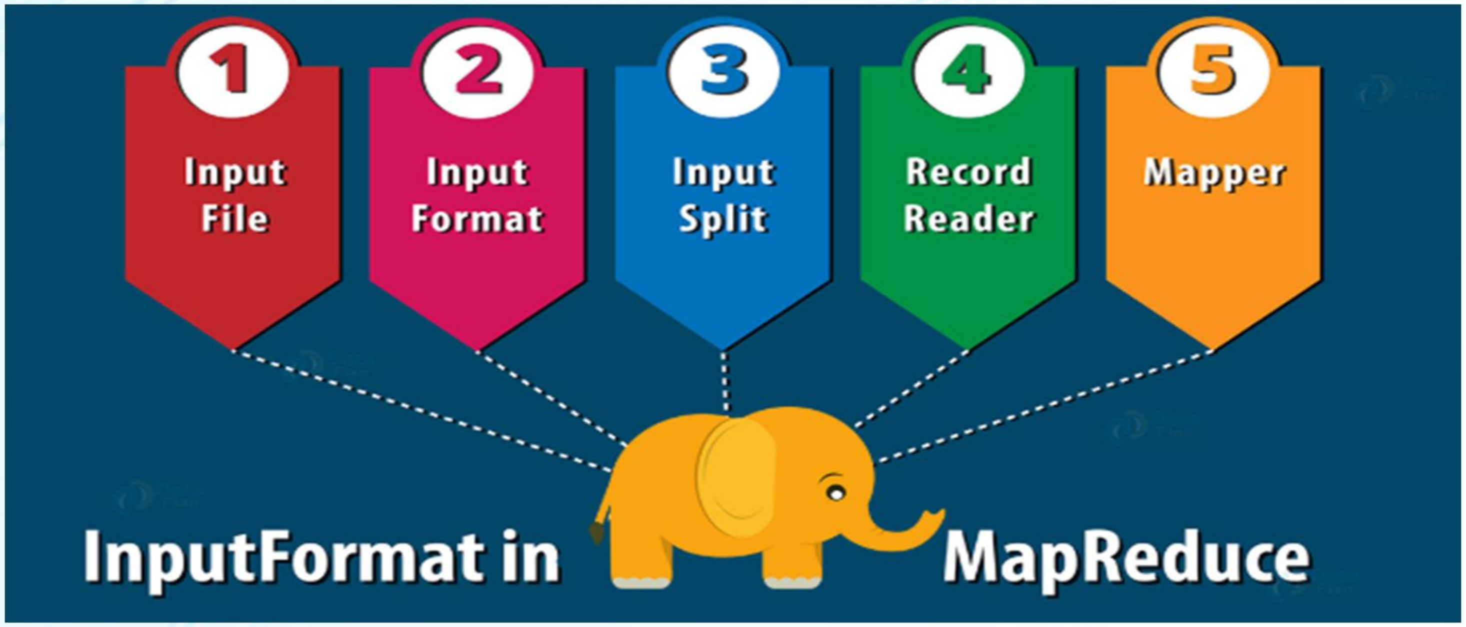
- Override this method for more complete control over the execution of the Mapper.

Phases of a Reducer

- Reducer has 3 primary phases:
  - Shuffle
    - The Reducer copies the sorted output from each Mapper across the network.
  - Sort
    - The framework merge sorts Reducer inputs by keys.
    - The shuffle and sort phases occur simultaneously i.e. while outputs are being fetched they are merged.
  - Reduce
    - In this phase the reduce(Object, Iterable, Context) method is called for each <key, (collection of values)> in the sorted inputs.
    - The output of the reduce task is typically written to a RecordWriter via TaskInputOutputContext.write(Object, Object).
- The output of the Reducer is not re-sorted.

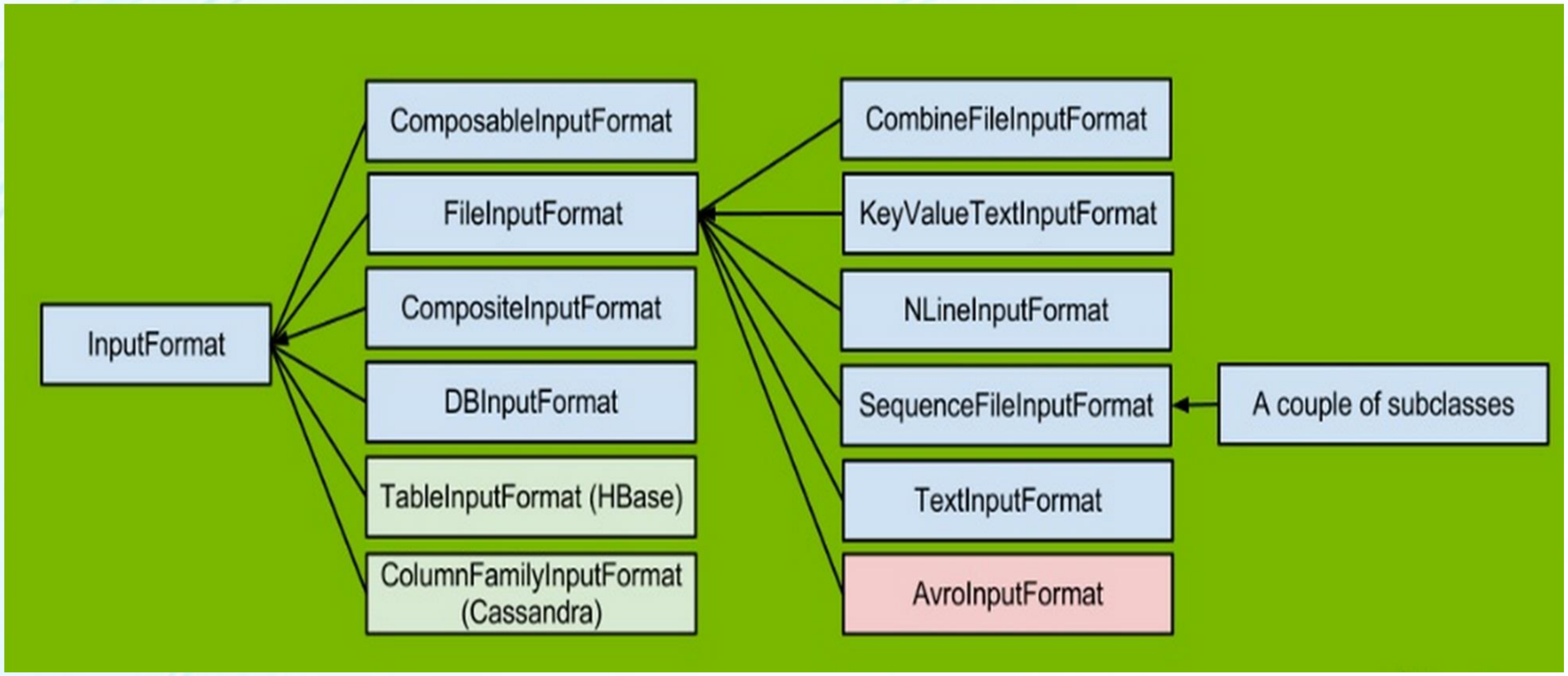
A few more details about MapReduce

InputFormat



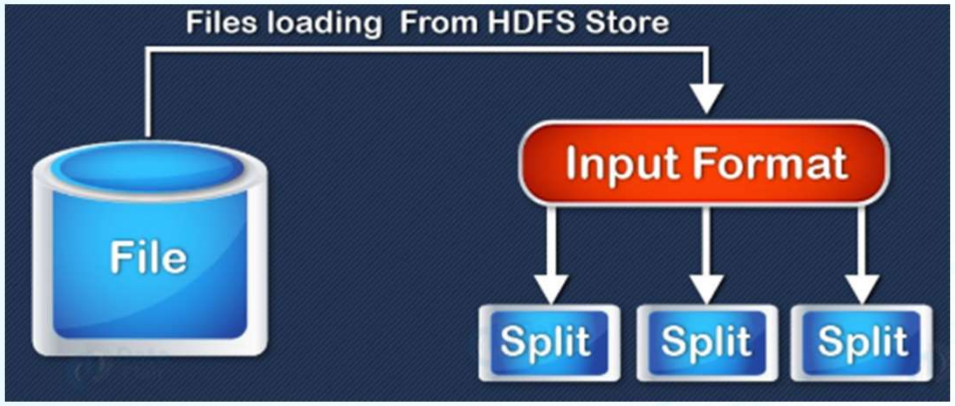
- InputFormat defines how input files are split up and read in Hadoop.
- An Hadoop InputFormat is the first component in Map-Reduce. It is responsible for creating the input splits and dividing them into records.
- InputFormat defines the RecordReader, which is responsible for reading actual records from the input files.

InputFormats

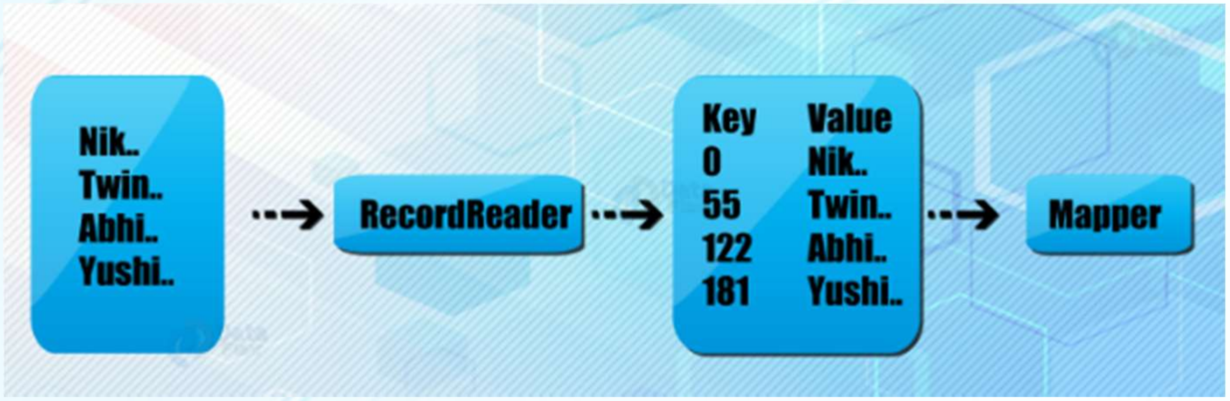


InputSplit

- Hadoop **InputSplit** represents the data which is processed by an individual Mapper. The split is divided into records. Hence, the mapper process each record (which is a key-value pair).
- Inputsplit does not contain the input data; it is just a reference to the data.
- **InputFormat** creates the Inputsplit and divide into records.
- You can alter InputSplit size using *`mapred.min.split.size`* config parameter



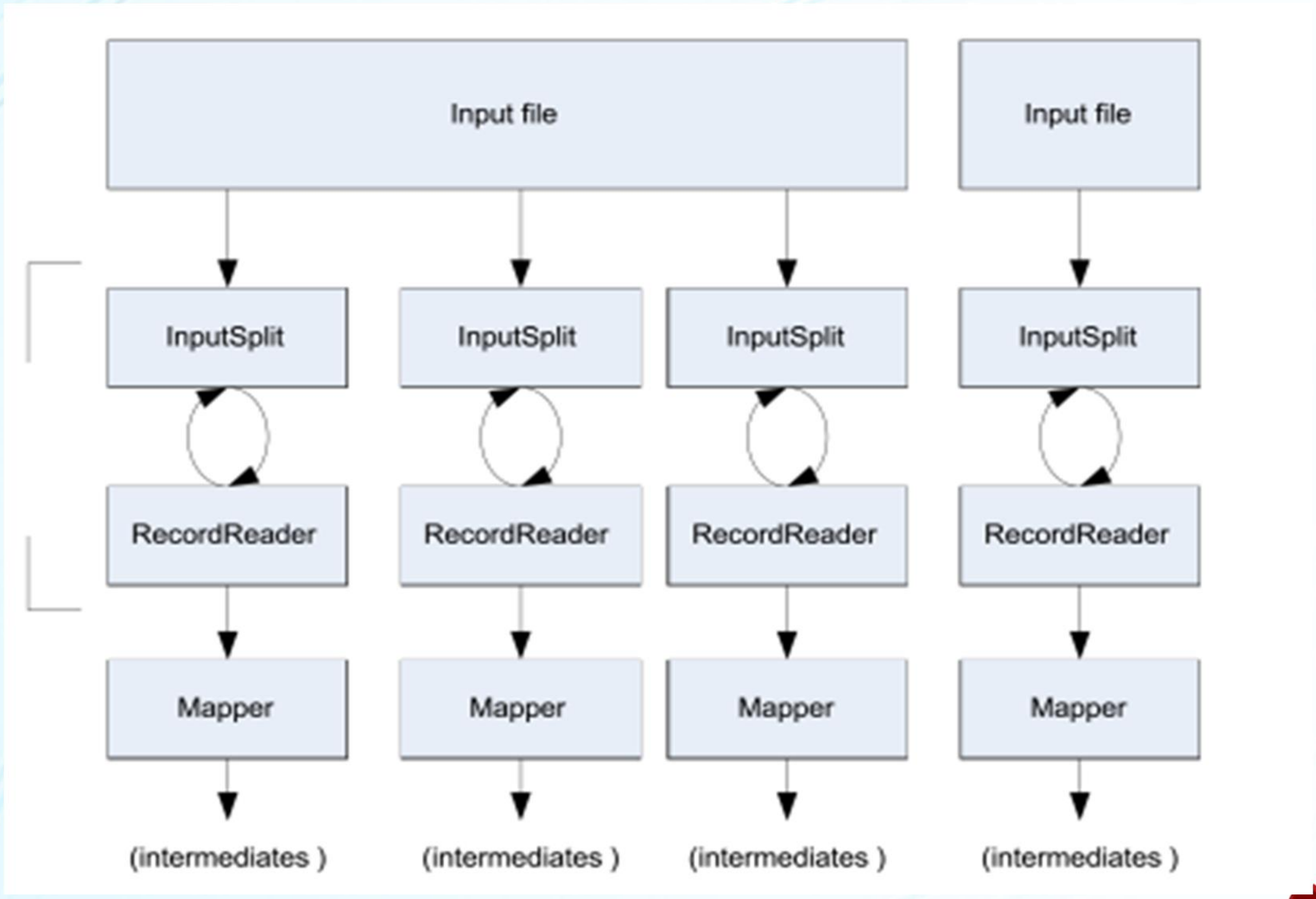
RecordReader



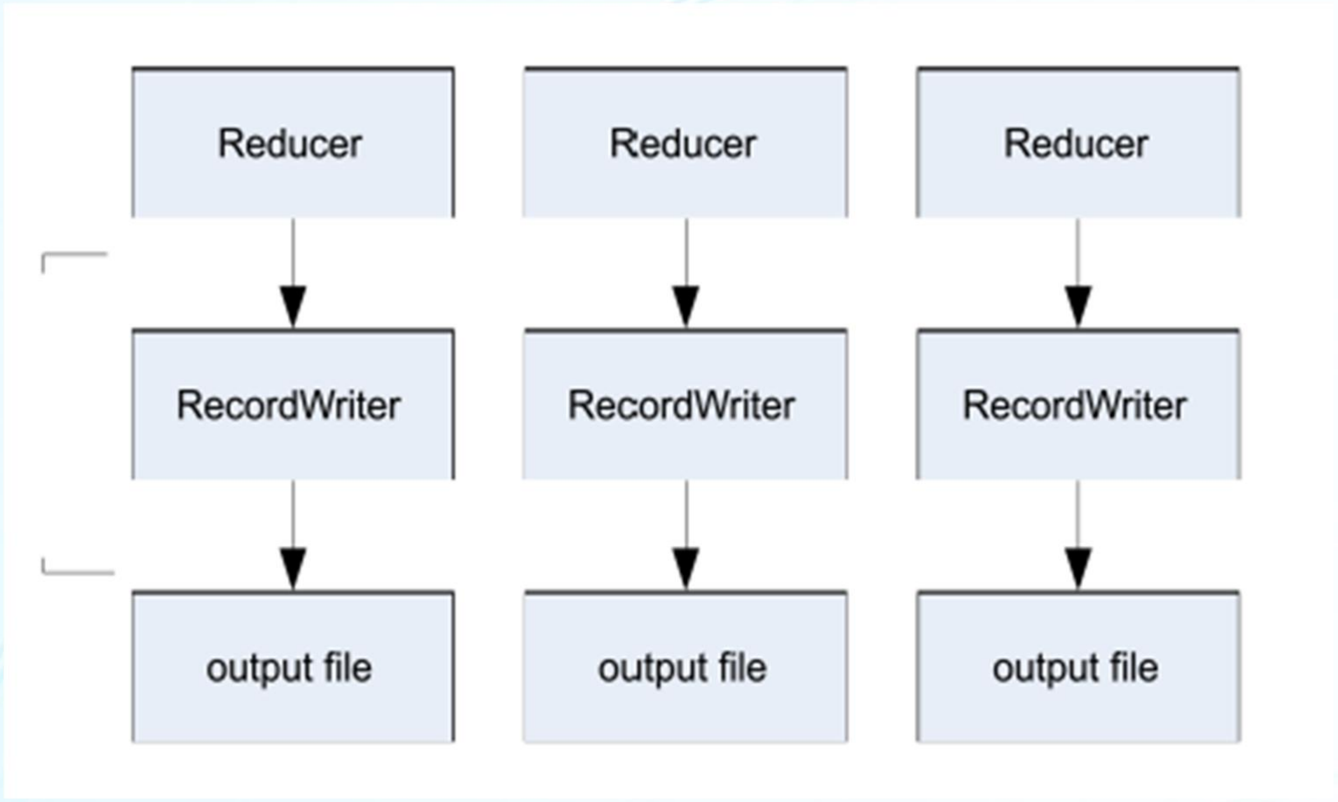
- Hadoop RecordReader uses the data within the boundaries that are being created by the InputSplit and creates Key-value pairs for the mapper.

- It communicates with the inputsplit until the file reading is not completed.

Getting data to the Mapper



Writing output from the Reducer



Writable Data Types

- Hadoop has its own data types called Writable data types that are meant for writing the data to the local disk and it is a serialization format.
- These Writable data types are passed as parameters (input and output key-value pairs) for the mapper and reducer.
- The Writable data types implements **WritableComparable** interface.
- Comparable interface is used for comparing when the reducer sorts the keys, and Writable can write the result to the local disk.
- MapReduce does not use the java Serializable because java Serializable is too big or too heavy for Hadoop. Writable can serialize the hadoop object in a very light way.

Writable Data Types

Java	Hadoop
String	Text
Int	IntWritable
Long	LongWritable
Float	FloatWritable
Double	DoubleWritable
Byte	ByteWritable
Null	NullWritable

GenericOptionsParser

- `GenericOptionsParser` is a utility class with in **`org.apache.hadoop.util`** package.

- This class parses the standard command line arguments and sets them on a configuration object which can then be used with in the application.

- The way to use `GenericOptionsParser` class is to **implement** `Tool` **interface** and then **use** `ToolRunner` to run your application.

- Options that are supported by `ToolRunner` through `GenericOptionsParser` are:

  - `-conf <configuration file>`
  - `-D <property>=<value>`
  - `-fs <file:///> or <hdfs://namenode:port>`
  - `-files <comma separated list of files>`
  - `-libjars <comma seperated list of jars>`

Tool interface

- `Tool` interface supports handling of generic command-line options.
- The tool/application should delegate the handling of standard command-line options to `ToolRunner.run(Tool, String[])` and only handle its custom arguments.
- Do not call the Tool's run method directly. Call `ToolRunner`'s static `run()` method, which takes care of creating a `Configuration` object for the tool before calling it's run method.

ToolRunner Class

- `ToolRunner` is a utility to help run classes implementing `Tool` interface.
- It works in conjunction with `GenericOptionsParser` to parse the generic hadoop command line arguments and modifies the Configuration of the `Tool`.
- The application-specific options are passed along without being modified.

THANK YOU